WebForms Page Framework

Author	Scott Guthrie
Area	XSP
Feature	WebForms Page Framework
Program Management Contact	ScottGu
Development Contact	DavidEbb
Test Contact	CharlesZ
Current Status	In Progress
External Reviewers	GaryBu, DmitryR, Smillet, ChrisAn, EricZ, BradLo

1. Overview

The ASP+ Page Framework - aka "WebForms" -- is a scalable COM+ Runtime Programming Model that can be used on the server to dynamically generate web pages.

Intended as a logical evolution of ASP (we provide syntax compatibility with existing pages), the WebForms Page Framework has been specifically designed to address a number of key deficiencies with the previous model. In particular: 1) The ability to create and consume reusable UI controls that can encapsulate common functionality and dramatically reduce the amount of code a page developer has to author. 2) The ability for development/authoring tools to provide strong WYSIWYG design support for pages (existing ASP code today is opaque to a tool). 3) The ability for developers to easily structure their page logic in an orderly -- non-"spaghetti code" - fashion.

This specification describes how we intend to address each of the above issues with WebForms, and details the core classes, interfaces and execution model exposed by the resulting framework.

1.1 Goals

- Maintain syntax and runtime compatibility with legacy ASP pages. Provide a solution that allows ASP Developers to rename their existing files to ".aspx" without having to make significant source code changes. Enable these developers to incrementally add declarative server controls to these pages, and gracefully take advantage of new WebForms functionality.

- Provide a new hierarchical page containership model that allows developers to create and consume UI controls that can be composed on the server. Enable them to utilize "nested inline templates" to provide rich customization of their look and feel. Ensure that they can be seamlessly edited with a WYSIWYG tool designer (specifically VB7 and FrontPage).

- Promote a page-processing model that enables page and control developers to intelligently participate in the servicing of incoming web requests. Enable developers to write easily maintainable code that provides logical execution structure. Allow developers to avoid the "spaghetti-code" problems prevalent in so many ASP applications today.

- Remove the current ASP dependency on interpreted script engines for runtime execution. Ensure that all "script code" written using either VBScript or JScript is dynamically compiled into COM+ IL -- and then JIT'd into native code by COM+ -- for dramatically improved page execution.

1.2 Aliases

For additional information or discussions about the WebForms Page Framework please join the ***XSP Web Forms*** (xspforms) or ***XSP Discussions*** (xsp) aliases.

1.3 High-Level Example

The below sample demonstrates a simple WebForms Page that prompts a customer for an AccountID and then displays their appropriate stock portfolio information:

```
<%@ Page Description="Simple Sample Page" Errorpage="ErrorPage.aspx" %>

<html>
    <script language="VB" runat=server>

        Sub AccountBtn_Click(Source as Object, E as EventArgs)

            If (AccountID.Text <> "") Then
                MyGridControl.QueryParams(0) = AccountID.Text
            End If

        End Sub

    </script>

    <body>

        <form action="SimplePage.aspx" method="post" runat="server">

            AccountID: <input id="AccountID" type=text runat=server>
                       <input type=button onServerclick="AccountBtn_Click" runat=server>

            <acme:gridcontrol id="MyGridControl"
                              query="select * from investments where AccountID='?'"
                              datasource="dsn:investmentserver"
                              runat="server">

                <template name="headertemplate">
                    <font face="verdana" color="red">
                        <b><%# Container.DataItem.Name %></b>
                    </font>
                </template>

                <template name="gridcelltemplate">
                    <font face="verdana" color="blue">
                        <%# Container.DataItem.Value %>
                    </font>
                </template>

            </acme:gridcontrol>

        </form>

    </body>
</html>
```

The following sections of this specification detail how the above file is parsed and dynamically compiled into a COM+ class at server runtime, as well as the execution sequence that occurs when an instance of the page is used to process an incoming web request.

2. WebForms Page Syntax

A *Page* is a declarative text file that contains markup syntax for coding server-side page logic, and composing dynamic output content (typically HTML or XML). At runtime it is parsed and compiled by the *Page Compiler* into a dynamically generated COM+ class. This class is then cached and used to process each incoming web request for that particular URL.

Page Files are composed from the following different syntax/semantic elements:

Page Directives:

Page directives specify optional settings used by the page-compiler when processing .aspx files (encoding, transaction semantics, session state requirements, etc).

Code Declarations:

Code declaration blocks define member variables and methods that will be compiled into the generated Page class. Developers use these blocks to author page/navigation logic.

Code Render Blocks:

Code Render Blocks enable developers to inline rendering code within their HTML content. This code is evaluated at page "render-time" to assist in generating final page output.

Server-Side Comments:

Server-Side Comments enable page developers to prevent server code (including server controls) and static content from executing/rendering.

HTML Server Controls:

HTML Server controls enable page developers to programmatically manipulate HTML elements within a page.

Custom Server Controls:

Custom server controls enable page developers to dynamically generate HTML UI and respond to client requests. They are represented within a file using a declarative, tag-based, syntax.

Hierarchical DataBinding Expressions:

WebForm's built-in DataBinding support enables page developers to hierarchically bind control properties to data container values.

Object Tag Declarations:

Object Tags enable page developers to declare and instantiate variables using a declarative, tag-based, syntax.

Server Side Includes:

Server-Side #Includes enable developers to insert the raw contents of a specified file anywhere within an XSP Page.

The following nine sections of this specification discuss each of the above syntactical/semantic elements in detail, and provide examples illustrating their use.

2.1 Directive Syntax

Page directives specify optional settings used by the Page Compiler when processing files. They are located at the top of a page file, and have the syntax:

```
<%@ directive {attribute=value}* %>
```

WebForms supports five directives: "page", "implements", "import", "register", "assembly" and "outputcache". Each directive supports one or more *case-insensitive* attribute/value pairs.

Page Directive

The "Page" directive defines a number of page specific attributes that the Page Compiler uses when compiling a new page class. These include (note: default values underlined where appropriate):

Attribute	Supported Values	Description
AUTOEVENTWIREUP	True, False	Indicates whether pages events autowired
BUFFER	True, False	Indicates response buffering semantics
CONTENTTYPE	Any content-type string	Indicates HTTP Content-Type of Output
CODEPAGE	Any valid codepage value	Indicates locale codepage of the file
CULTURE	Any COM+ culture string	Indicates culture setting of page
DEBUG	True, False	Whether compiled w/ debug symbols
RESPONSEENCODING	Encoding.GetEcoding()	Response encoding of content
LCID	Any valid LCID indentifier	Indicates locale identifier for code
DESCRIPTION	Any string description	Provides text description of page
ENABLESESSIONSTATE	True, False, ReadOnly	Indicates session state requirements
ERRORPAGE	Any valid HTTP URL	Redirect URL for un-handled errors
INHERITS	Page-derived COM+ class	Code-behind class for page to inherit
LANGUAGE	VB, JavaScript, C#	Programming language used within page
MAINTAINSTATE	True, False	Indicates if page viewstate is maintained
TRANSACTION	NotSupported, Supported, Required, RequiresNew	Indicates the transaction semantics
TRACE	True, False	Indicates whether tracing is enabled
SRC	Source file name	Code-behind class to compile
WARNINGS	True, False	Are compiler warnings errors or ignored
EXPLICIT	True, False	Indicates VB Option Explicit Mode
STRICT	True, False	Indicates VB Option Strict Mode

```
<%@ Page Language="VB" Transaction="Required" ContentType="text/xml" %>
```

It instructs the Page Compiler to use VB as the inline code language engine, mark the resulting COM+ class with metadata indicating that it requires a wrapping transaction, and set the default HTTP Mime ContentType transmitted to the client to be "text/xml".

Import

The "Import" directive allows page developers to "import" a COM+ namespace into a page - making all classes and interfaces within it available to a programmer. The following attributes/values are supported by the "import" directive:

Attribute	Supported Values	Description
Namespace	Any valid COM+ Namespace	COM+ Namespace to Import

An example page that uses the "import" directive can be found below:

```
<%@ Import Namespace="System.IO" %>
<%@ Import Namespace="System.Net" %>

<html>
        <script language="VB" runat=server>

                Sub EnterBtn_Click(ByVal Source as Object, ByVal E as EventArgs)

                        MyFiles.DataSource = System.Array.AsList(File.GetFiles(Directory.Text))
                        MyFiles.DataBind()

                End Sub

        </script>

        <body>
                <form Action="DirectoryPage.aspx" method="POST" runat=server>

                        Directory to Browse: <input id="Directory" type=text runat=server>
                                             <button onClick="EnterBtn_Click" runat=server>

                        <asp:repeater id="MyDirectoryList" runat=server>

                                <template name="itemtemplate">
                                    <%# Container.DataItem %><br>
                                </template>

                        </asp:repeater>
                </form>
        </body>
</html>
```

Note that the following COM+ namespaces are *implicitly* imported into all pages automatically:

System, System.Web, System.Web.UI, System.Web.UI.HtmlControl, System.Web.UI.WebControls, System.Collections, System.Text, and System.IO.

Register

The "Register" directive allows page developers to register a tag prefix alias with a COM+ namespace – or alternatively "register" a single "tagprefix" + "tagname" combination with a user control. The Page framework will use this directive to control which instance of a server control is created by the parser at runtime.

Attribute	Supported Values	Description
TagPrefix	Any valid string	Tag prefix name to associate w/ control
TagName	Any valid string	Tag name to associate w/ control
Namespace	Any valid COM+ namespace	Namespace to associate tagprefix alias with
Src	Any local URL or file Path	Path to declarative control to instantiate

An example page that uses the "Register" directive with UI server controls can be found below. It registers the "Scottgu" tag prefix with the "Microsoft.Scottgu" namespace. It then registers the "Acme:Adrotator" tag with the "Adrotator.aspc" declarative control. At runtime the "scottgu: gridcontrol" tag will be handled by the Microsoft.ScottGu.GridControl class, and the "Acme:AdRotator" tag will be handled by the dynamic class generated from the Adrotator.aspc file.

```
<%@ Register TagPrefix="scottgu" namespace="Microsoft.ScottGu" %>
<%@ Register TagPrefix="Acme" TagName="Adrotator" Src="Adrotator.aspc" %>

<html>
        <script language="VB" runat=server>

                Sub AccountBtn_Click(ByVal Source as Object, ByVal E as EventArgs)

                    If (AccountID.Text <> "") Then
                        MyGridControl.Parameters(0) = AccountID.Text
                    End If

                End Sub

        </script>

        <body>
                <form action="Page1.aspx" method="POST" runat="server">

                    <acme:adrotator file="myAds.xml" category="money" runat="server"/> <br>

                    Please Enter AccountID:  <input id="AccountID" type=text runat=server>
                                             <button onclick="AccountBtn_Click" runat=server>

                    <scottgu:gridcontrol id="MyGridControl"
                                         query="select * from stocks where AccountID='?'"
                                         datasource="dsn:investmentserver"
                                         runat="server"
                    </scottgu:gridcontrol>
                </form>
        </body>
</html>
```

Note: In addition to configuring assembly controls/namespace imports at the page level, WebForms also allows developers to specify these at the application level (which are then automatically added to each page). A number of well-known packages will automatically be configured here out of the box.

Assembly

The "Assembly" directive allows page developers to declaratively link a COM+ assembly against the current page at parse-time. Alternatively, page developers can register such assemblies within the config.cfg file to link assemblies across an entire app. The following attributes/values are supported by the "Assembly" directive:

Attribute	Supported Values	Description
Name	Any valid assembly name	Assembly name to link against

An example page that uses the "assembly" directive can be found below. It indicates that the page should import the MyAssembly.Foobar.Baz and MyAssembly.Foobar.Baz namespaces and link against the MyAssembly.dll assembly.

```
<%@ Import Namespace="MyAssembly.Foobar.Baz" %>
<%@ Import Namespace="MyAssembly.Foobar.Biz" %>
<%@ Assembly Name="MyAssembly.dll" %>

<html>
        <body>
                Time is now: <%= Now %>
        </body>
</html>
```

OutputCache

The "OutputCache" directive allows page developers to declaratively control the server-side output caching policies of a page. The following attributes/values are supported by the "OutputCache" directive:

Attribute	Supported Values	Description
Duration	Any non-negative number	Number of seconds to output cache page
Vary	Semi-colon separated header list	HTTP Headers to vary output by

Todo: ErikOls will add more support to this directive in the future.

An example page that uses the "outputcache" directive can be found below. It indicates that the page (which prints out the current time) should be output cached for 60 seconds at a time:

```
<%@ OutputCache Duration="60" %>

<html>
        <body>
                Time is now: <%= Now %>
        </body>
</html>
```

Implements

The "Implements" directive allows page developers to declaratively indicate that a page implements a given COM+ interface. The following attributes/values are supported by the "OutputCache" directive:

Attribute	Supported Values	Description
Interface	Any valid interface name	Name of interface to implement

An example page that uses the "implements" directive can be found below. It indicates that the page implements the IfooBar interface:

```
<%@ Implements Interface="MyNamespace.IFooBar" %>

<html>
        <script language="C#" runat=server>

                void FoobarMethod() {

                }

        </script>

        <body>
                Time is now: <%= Now %>
        </body>
</html>
```

2.2 Code Declaration Syntax

Pages support two types of code blocks: 1) code declaration blocks that define page member variables and methods, 2) code render blocks responsible for executing and optionally generating output during page rendering.

Code declaration blocks are defined within a page using <script> tags that contain a "runat" attribute whose value is defined as "server". The script tag may optionally utilize a "language" attribute to specify the syntax of its inner code. If none is specified, WebForms will default to the language configured for the base page (controlled using a page declarative). The script tag may also optionally specify an external file to load code from via a "src" attribute. Note that all inner content within a <script runat=server> tag is ignored if an external source file is specified.

Developers can define any method within a code declaration block. WebForms provides special recognition of six methods named:

- Page_Init
- Page_Load
- Page_DataBind
- Page_PreRender
- Page_Unload

These methods are treated as event handlers for the standard events exposed off of the page class. The WebForms Page Framework will automatically wire-up appropriate delegate instances at runtime for these methods – saving page developers from having to write the necessary "glue-code".

Syntax:

```
<script runat="server" language="language" src="externalfile">
        Code Goes Here....
</script>
```

Example:

The below example demonstrates how a <script runat=server> block can be used to implement the Page_Load() event.

```
<html>
        <script language="VB" runat="server" >

                Sub Page_Load(Sender as Object, E as EventArgs)
                        Message.Text = "Hi Scott, Welcome to ASP+!"
                End Sub

        </script>

        <body>
                Here is a message: <span id="message" runat="server"/>
        </body>
</html>
```

2.3 Code Render Block Syntax

Code Rendering Blocks are defined using <% %> or <%= %> syntax fragments, and are logically combined into a single "rendering" method that executes at page render time (more details regarding this "logical combination" are provided later in the spec).

Syntax:

Inline code (can be used to define either self-contained code blocks or control flow blocks):

```
<% Inline Code %>
```

Inline expressions (short-cut for calling Response.Write on returned code value):

```
<%= Inline Code %>
```

Note: The expression contained within a <%= expr %> block is ultimately wrapped by a call to "Response.Write(expr)" – as such developers using C# should not use a semi-colon at the end of the expression (it will produce a compile-time error if you do).

Example:

The below example demonstrates code-rendering blocks that output values and provide control flow (note: developers will lose tool WYSIWYG support when doing things like this):

```
<html>
    <script language="VB" runat="server">

        Dim Name as String
        Dim Age as String

        Sub Page_Load(Sender as Object, E as EventArgs)
            Name = Request.Querystring("Name")
            Age = Request.Querystring("Age")
        End Sub

    </script>

    <body>
        <% If (Name <> "") Then %>

            Here is your message:

            <% For x=1 to 7 %>
                <font size="<%=x%>"> Hi <%=Name%>, you are <%=Age%>!</font>
            <% Next x %>

        <% End If %>
    </body>
</html>
```

2.4 Server-Side Comment Block Syntax

Server-Side Comment Blocks are defined within <%-- --%> fragments, and enable page developers to prevent server code (including server controls) from executing, and template content from outputting (the Page Parser ignores everything within them).

Syntax:

```
<%-- Commented out code/content --%>
```

Note: Page developers can also use their native language comment semantics within <script runat=server></script> and <% %> code blocks.

Example:

The below example demonstrates the use of a Server-Side Comment Block:

```
<%@ Language="VB" Description="Example demonstrating server-side comments" %>

<html>
    <script language="VB" runat="server">

        Dim Name as String
        Dim Age as String

        Sub Page_Load(Sender as Object, E as EventArgs)

            Name = Request.Querystring("Name")
            Age = Request.Querystring("Age")

        End Sub

    </script>

    <body>
        <%--
            <asp:adrotator file="myAds.xml" runat="server"/>

            <% If (Name <> "") Then %>

                Here is your message:

                <% For x=1 to 7 %>
                    <font size="<%=x%>">
                        Hi <%=Name%>, you are <%=Age%>!
                    </font>
                <% Next x %>

            <% End If %>
        --%>
    </body>
</html>
```

Note: It is illegal to embed <%-- --%> blocks within open <% %> blocks. For example, the following sample code: <% <%-- comment --%> %> will not work.

2.5 HTML Server Control Syntax

By default, all HTML tags within a page are treated as literal text content and are programmatically inaccessible to page developers. Page authors can indicate that an HTML tag should be parsed and treated as an accessible server control by marking it with a "runat" attribute whose value is set to "server". They may optionally specify a unique "id" attribute to enable programmatic referencing of the control, and can specify property arguments and event bindings on server control instances using declarative name/value attribute pairs on the tag element.

HTML Control Syntax:

```
<HTMLTag id="OptionalName" runat=server>
</HTMLTag>
```

Note: While the page parser does not require literal – non-dynamic – HTML content to be "well formed", it does require that all non-closed HTML tags be properly closed (either with a trailing "/" or an end tag) and must be cleanly nested (overlapping tags are not allowed).

HTML Control Example:

The below example demonstrates the use of an HTML text-box, drop-down, button and span control:

```
<html>
    <script language="VB" runat=server>

        Sub MyButton_Click(ByVal Source as Object, ByVal E as EventArgs)
            MyMessage.InnerHtml = MyName.Value & " is a " & MyTitle.Value
        End Sub

    </script>

    <body>
        <form action="MyPage.aspx" method="POST" runat=server>

            Name:       <input type="Text" id="MyName" runat=server>

            Occupation: <select id="MyTitle" size=1 runat=server>
                            <option>Software Design Engineer</option>
                            <option>Software Test Engineer</option>
                            <option>Software Design Engineer/Test</option>
                            <option>Program Manager</option>
                        </select>

            <button OnServerClick="MyButton_Click" runat=server>
                Enter
            </button>

            <br><br>

            <span id="MyMessage" runat=server> </span>

        </form>
    </body>
```

```
</html>
```

Supported HTML Controls

WebForms provides built-in support for the following HTML Control Tags (please review the Html Server Controls Spec on http://xsp for more details):

HTML Tag Name	Example	COM+ Class
<a>	 My Link 	HtmlAnchor
<img>	<img id="MyImage" runat=server>	HtmlImage
<form>	<form id="MyForm" runat=server> </form>	FormControl
<select>	<select id="MyList" runat=server> <option>One</option> <option>Two</option> </select>	HtmlSelectList
<input type=file>	<input id="MyFile" type=file runat=server>	HtmlInputFile
<input type=text>	<input id="MyTextBox" type=text>	HtmlInputText
<input type=password>	<input id="MyPassword" type=password>	HtmlInputText
<input type=reset>	<input id="MyReset" type=reset>	HtmlInputButton
<input type=radio>	<input id="MyRadio" type=radio runat=server>	HtmlInputRadio
<input type=checkbox>	<input id="MyChk" type=checkbox runat=server>	HtmlInputCheckbox
<input type=hidden>	<input id="MyHid" type=hidden runat=server>	HtmlInputHidden
<input type=image>	<input type=image src="foo.jpg" runat=server>	HtmlInputImage
<input type=submit>	<input type=submit runat=server>	HtmlInputButton
<input type=button>	<input type=button runat=server>	HtmlInputButton
<button>	<button id=MyButton runat=server>	HtmlButton
<textarea>	<textarea id="MyText" runat=server> This is some sample text </textarea>	HtmlTextArea
<table>	<table id="MyTable" runat=server> <tr> <td>Text</td> </tr> </table>	HtmlTable

Note: WebForms will use the generic "HtmlGenericControl" class to handle scenarios when an HTML Tag that is not listed above is marked with a "runat=server" attribute pair.

2.6 Custom Server Control Syntax

Custom Server Controls enable developers to create reusable components that encapsulate common programmatic functionality. Page developers utilize server controls by specifying declarative tags within their page file. These tags must have a "runat" attribute whose value is set to "server". They may optionally specify a unique "id" attribute to enable programmatic referencing of the control.

Developers can optionally specify property arguments and event bindings on a server control instance using declarative name/value attribute pairs on its tag element. They can also specify in-line template parameters to server controls (for complex properties) by providing an appropriate "template" prefixed child-element to the parent server control.

Server Control Declaration Syntax:

Developers declare a custom server control within a Page using an XML/HTML tag that contains a "runat" attribute whose value is set to "server". WebForms will use the tag's "element name" to resolve the actual COM+ class to instantiate and use when processing a request (this reference is declared using a "reference" directive - as described in 2.1 of the spec).

```
<TagPrefix:TagName id="OptionalId" runat=server />

        or

<TagPrefix:TagName id="OptionalId" runat=server />  </TagPrefix:TagName>
```

Server Control Property Syntax:

Developers specify properties on server controls using a declarative attribute name/value pair syntax (where the case-insensitive attribute name represents the property name and the value represents the property value to assign).

```
<TagPrefix:TagName id="OptionalId" propertyname="propertyvalue" runat=server />
```

Developers can declaratively set all of the standard COM+ "primitive" datatypes directly:

Supported Property DataTypes			
String	Integer	Double	Byte
Char	Boolean	DateTime	TimeSpan
Decimal	Variant	Currency	

For example:

```
<ASP:image id="myimage" imageurl="foo.gif" runat="server"/>

        or

<ASP:textbox id="mytextbox" maxlength="55" runat="server"/>
```

Developers can also specify properties represented using Enums (where the parser will treat the attribute value of the control as the enum value name).

```
<ASP:image imageurl="foo.gif" ImageAlignment="Left" runat=server/>
```

In addition to setting properties on controls, developers can also declaratively set properties on the properties of controls using a name/attribute pair syntax where sub-properties are identified via "-" characters).

```
<TagPrefix:Classname propertyname-subpropertyname="propertyvalue" runat=server />
```

For example, the syntax below demonstrates how to set the Font.Name property on the Label server control:

```
<ASP:Label font-name="Arial" runat=server>
        Four score and seven years ago....
</ASP:Label>
```

Server Control Event Wiring Syntax:

Developers specify event wirings on server controls using a declarative attribute name/value pair syntax (where the attribute name represents the control's event name and the value represents the method on the Page that should be called when the event is fired).

```
<TagPrefix:Classname EventName="EventHandlerMethodName" runat=server />
```

At runtime, WebForms will dynamically wire up the appropriate -- type-safe -- COM+ delegate object between the control event "adder method" and target page method (note: the target method can either be defined declaratively within a <script runat=server></script> block or in the code behind file.

For example, the below declarative syntax would automatically wireup the "OnClick" event on a button control:

```
<html>
        <script language="VB" runat=server>

                Sub MyButton_Click (Sender as Object, Evt as EventArgs)
                        ' Do something when someone clicks the button
                End Sub

        </script>

        <body>
                <form action="default.aspx" method="post" runat=server>

                        <ASP:Button text="Hi" OnClick="MyButton_Click" runat=server/>

                </form>
        </body>
</html>
```

Server Control In-Line Template Syntax:

In-line template properties enable *control developers* to build controls that are as "lookless" as possible – abstracting their behavior away from their actual appearance. *Control consumers* can take advantage of this functionality to provide substitutable template parameters that enable rich – and hierarchical – customization.

Template properties are specified on a server control by declaring "template" sub-element children on a parent server control. The "name" attribute of these "template" sub-elements identifies a property name on the parent control whose type is of "ITemplate".

```
<TagPrefix:Classname id="OptionalName" runat=server>

        <template name="templatepropertyname">
                Inline Template Value
        </template>

</TagPrefix:Classname>
```

WebForms interprets this syntax at parse-time to dynamically bind an appropriate ITemplate instance to a property of the same name on the parent server control. The server control can then utilize this ITemplate instance at runtime to create and initialize 0 to N new instances of the appropriate parameter control class.

For example, the below example demonstrates how a page developer could customize the individual item templates on a repeater control:

```
<ASP:Repeater id="myrepeater" runat=server>

        <template name="headertemplate">
                This is the header....
        </template>

        <template name="itemtemplate">
                This is an item in the template
        </template>

</ASP:Repeater>
```

2.7 DataBinding Expression Syntax

WebForms supports a hierarchical DataBinding model that allows page developers to declaratively associate binding expressions between server control properties and data sources. WebForms allows any server control property to be databound (the property is specified declaratively as a attribute name on the server control). Controls can DataBind against any public field or property on either the page or their immediate naming container (please review the "DataBinding" section of this specification for full details).

Syntax:

DataBinding expressions are defined using <%# %> syntax fragments that can be written either on the "value" side of a property definition:

```
<servercontrol propertyname="<%# DataBinding Expression %>" runat=server/>
```

or inline within a literal text:

```
This the value you databound: <%# DataBinding Expression %>
```

Example:

The below example demonstrates the use of DataBinding with a template-based list control (note: the repeater control is databound to fields on the page, and the spans are databound to the repeater's item containers):

```
<html>
        <script language="VB" runat=server>

                Sub Page_Load(Sender as Object, E as EventArgs)
                    MyOrderSystem as New OrderSystem
                    MyRepeater.DataSource = MyOrderSystem.GetOrdersFromDay(Now)
                    MyRepeater.DataBind()
                End Sub

        </script>

        <body>
                <form action="DatabondPage.aspx" method="post" runat=server>

                    <asp:repeater id="MyRepeater" runat=server>

                        <template name="itemtemplate">
                                <img src="<%# Container.DataItem.ImageUrl %>" runat=server>
                                <b> Description: </b> <%# Container.DataItem.Description %>
                        </template>

                    </asp:repeater>

                </form>
        </body>
</html>
```

2.8 Server-Side Object Tag Syntax

Server-Side Object Tags enable page developers to declare and instantiate page variables using a declarative, tag-based, syntax.

When the Page Parser/Compiler encounters a server-side object tag declaration within a Page File, it automatically generates a get/set property on the page (using the "id" attribute of the tag as the property name). The "get" property accessor is then wired-up to create an instance of the object on first use. Note that the resulting instance is not added as an object within the page's hierarchical server control tree (it is instead treated as a non-UI variable declaration).

Syntax:

Server-Side Object Tags can be used to create COM+ Runtime Classes, Classic COM ProgID Objects, and Classic COM CLSID Registered Components. The type of object to create is identified using three different tag attributes – "class", "progid", "classid". These attributes are mutually exclusive – it is an error to use more than one at any given time.

```
<object id="id" runat=server class="COM+ Class Name">
<object id="id" runat=server progid="Classic COM ProgID"/>
<object id="id" runat=server classid="Classic COM ClassID"/>
```

Attributes	Description
ID	Unique "name" to use when declaring object on page
Runat	Must be set to "server" for the object to execute within XSP. All non-server values cause the page compiler to assume that the object tag should be transmitted down for a client to handle.
Class	Identifies COM+ Class to Create
ProgID	Identifies Classic COM Component to Create
ClassID	Identifies Classic COM Component to Create
LateBinding	Indicates whether early/late binding wrappers should be used with classic com components that have been run through TLBIMP.exe (default is false)

Example:

```
<html>
        <object id="MyDatabase" class="Microsoft.WFC.XDO.OLEDBAdaptor" runat="server"/>

        <script language="VB" runat=server>

                Sub Page_Load(Sender as Object, E as EventArgs)
                        Dim StockDetails as Recordset
                        Set StockDetails = MyDatabase.Execute("DSN:money", "select * from stocks")
                End Sub

        </script>
</html>
```

2.9 Server-Side #Include Directive Syntax

The Server-Side #Include Directive enables developers to insert the raw contents of a specified file anywhere within a Page. It is processed before any dynamic code is executed (much like a C pre-processor). *Note: It must be surrounded by HTML/XML comment delimiters to avoid being interpreted as static text.*

Syntax:

```
<!-- #include PathType = FileName -->
```

Parameters:

PathType

Specifies the type of the path to *FileName*. The path type can be one of the following:

Path Type	Meaning
File	The file name is a relative path from the directory containing the document with the **#include** directive. The included file can be in the same directory or in a subdirectory; it cannot be in a directory above the page with the **#include** directive.
Virtual	The file name is a full virtual path from a virtual directory in your Web site.

FileName

Specifies the name of the file to be included. *FileName* must contain the file name extension, and you must enclose the file name in quotation marks (").

Example:

```
<html>
    <body>
        <!-- #Include file="header.inc" -->
        Here is the main body of my file:
        <% For I=0 to 10 %>
            <!-- #Include virtual="/Includes/Foobar.inc" -->
        <% Next %>
        <!-- #Include virtual="footer.inc" -->
    </body>
</html>
```

3.0 WebForms Control Framework

One of the principal design goals of the WebForms Page Framework has been to provide a rich infrastructure that enables developers to create and consume reusable UI controls that can encapsulate common functionality and dramatically reduce the amount of code a page developer has to author.

It is our contention that a fundamental requirement of any rich UI control framework is the ability for developers to build encapsulated controls that are capable of both generating rich UI views ***and*** responding to end-user's interaction/manipulation of them. This later feature is extremely tricky with the disconnected + stateless model of the web (to our knowledge no other competing web programming framework has attempted to provide this).

To make it both easy and possible for developers to accomplish this, the WebForms Page Framework delivers a rich runtime infrastructure that provides:

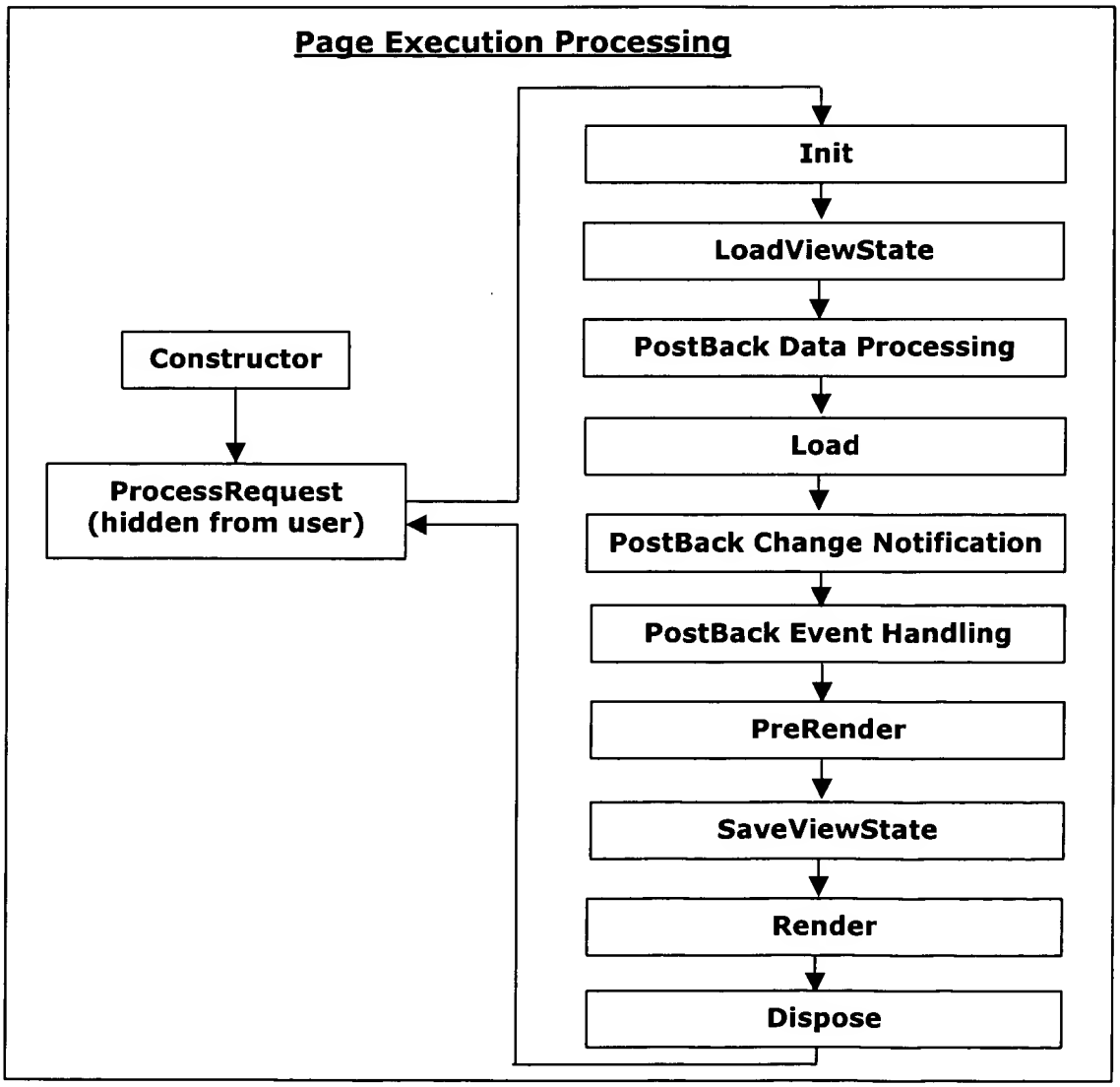
1) Event Based Execution Model
2) Tree Naming Services
3) ViewState Management Services
4) Post-Back Data Services
5) Post-Back Eventing Services
6) DataBinding Services
7) Template UI Customization Services

The following sections of this spec go into detail regarding each of these capabilities, as well as the page processing lifecycle that encapsulates them at runtime.

3.1 Control Processing Lifecycle

The base Page class implements the HTTP Runtime-defined IHTTPHandler interface and provides a finalized implementation of the "ProcessRequest" method. When an XSP Server receives a client request for an page, it creates an appropriate customized instance of the Page class, and then invokes its "IHTTPHandler::ProcessRequest" method.

The Page class instantiates and populates a tree of server control instances representing the appropriate declarative ".aspx" file on each incoming web request. Once the tree is created, the Page class kicks-off a "staged" execution sequence that enables both user code and the server controls to intelligently participate in the request processing and rendering of the page. This execution processing is captured within the "Control" base class contract and is defined as:



Important Note: The complexity of the above state diagram does not need to be understood by end-user page developers (or even most control authors). Page developers will instead typically work with two methods – "Load" and "Dispose" – in addition to authoring postback event-handlers ("onClick", "onChange", etc) on individual server-controls.

On each incoming page request the Page class – as well as each of its child server controls (and all of their child controls, recursively) – will execute the below execution steps:

Init

Controls can use Init to initialize any settings that will be needed during the lifetime of the incoming web request (for example: loading external template files, setting up event wirings, etc). ViewState information is not available for use.

LoadViewState

Control's ViewState information is automatically populated within the "*Control.ViewState*" collection (see section 4.3 for details). The LoadViewState methods provides control developers with an appropriate hook for them to utilize this collection (if necessary) to restore their internal state settings appropriately.

PostBack Data Processing

Controls use the PostBack Data Processing stage to process any incoming form data and update their object models/internal state appropriately (note that change notification events are not raised during this stage – instead this occurs later in the lifecycle execution). This enables server control authors to automatically (and accurately) reflect the state of the control on the client (for example: if a user typed "scott" into a textbox on the client, the textbox control on the server should also reflect the text value "scott" at the end of this stage of page execution).

Note: Please review the "Post-Back Services" section of this spec for more details.

Load

Pages/Controls use Load to perform any actions common to each request (for example: setting up a database query, updating a timer on a page, etc). By this point in the lifecycle, all of the server controls in the tree have been created and initialized, been populated with the appropriate viewstate values, and had all form controls populated with the appropriate client-side data values.

PostBack Change Notification

Controls fire appropriate change notification events during the PostBack Change Notification phase in response to form/control value changes that occurred on the client between the previous and current postbacks to a page.

For example, a textbox server control whose initial value was "bob" when a page was first rendered, would raise an "OnChange" event to all registered listeners if the value of the textbox came back as "fred" on the next postback.

PostBack Event Processing

Controls use the PostBack Event Processing phase to correctly handle client-actions that caused a page postback -- and raise appropriate events on the server.

For example: a button server control could handle a postback from the client in response to a user clicking it, and then raise an appropriate "OnClick" event on the server for a page developer to handle.

PreRender

Controls use PreRender to perform any last minute update operations that need to take place immediately before page/control state is saved and output rendered.

SaveViewState

Control's ViewState information is automatically persisted from the "*Control.ViewState*" collection (see section 4.3 for details) into a string object immediately after the SaveViewState stage of execution. Control developers can override the SaveViewState event to optionally modify the ViewState collection in preparation for this.

Render

Controls use Render to generate appropriate output for the browser client. This is accomplished through a hierarchical - top-down - tree walk of all server controls and embedded (<% %>) rendering code.

Dispose

Pages/Controls use Dispose to perform any final cleanup work (for example: closing files or database connections) before they are torn-down.

Note: In the new COM+ Runtime world, it is very important that expensive resources (like database connections) are explicitly closed. Otherwise, they will remain open until the next Garbage Collection (GC) occurs. On a heavily loaded server (which might process hundreds of requests between GCs), this can quickly cause resource exhaustion (for example: hundreds of open connections to a database server).

3.2 Tree Naming Services

A central necessity of any control architecture is the ability to uniquely name -- and subsequently locate – a given control on a page. WebForms accomplishes this by allowing developers to assign each server control on a page with an "ID" identifier that they can then use to later reference it (note: controls that are not given an "ID" value by a page developer are automatically assigned one by the page framework).

In order to better support hierarchical scenarios like repetition (where multiple templated controls that share the same ID might need to be created), the WebForms control framework supports the notion of "NamingContainers". NamingContainers are defined as controls that implement the INamingContainer marker interface, and logically define a new ID namespace hierarchy within the tree. A control's "ID" value is guaranteed to be unique among all controls that share its "NamingContainer" within the tree. A control's "UniqueID" property provides read-only access to the fully qualified hierarchical "ID" (composed with each naming container's ID concatenated before the control's local ID).

For example, the below sample demonstrates a simple repetition scenario that will cause a repeater control to add three "Message" controls within the tree.

```
<html>
    <script language="C#" runat=server>

        void Page_Load(Object sender, EventArgs e) {
            MyRepeater.Datasource = new String[3]{ "One", "Two", "Three" };
            MyRepeater.DataBind();
        }

    </script>

    <body>
        <form action="Repeater.aspx" method="Post" runat=server>

            <asp:repeater id="MyRepeater" runat=server>
                <template name="itemtemplate">
                    <span id="Message" runat=server>
                        <%# Container.DataItem %>
                    </span>
                </template>
            </asp:repeater>

        </form>
    </body>
</html>
```

Rather than add duplicate "Message" IDs into the root naming container of the tree, the Repeater – and its RepeaterRowItem control -- implements the INamingContainer interface. This ensures that ID values do not clash when the page is run, and that each "Message" control is cleanly segregated into a separate logical namespace within the tree.

As such, while the local "ID" property of each of the above span controls is "Message", the actual fully qualified UniqueID values of the controls are actually: "MyRepeater:Ctrl0:Message", "MyRepeater: Ctrl1:Message" and "MyRepeater:Ctrl2:Message".

A control's ID value is used heavily at runtime during the postback phase of page execution to route client data values and postback event references to appropriate server controls.

Developers can also utilize a control's "ID" value to obtain references to the server control instances at runtime (allowing them to program against them). Developers can search the tree for a given "ID" using the "FindControl" method exposed on the Control base class. The FindControl method will perform a relative search of the control's naming container for the given control, and will return a reference to it if found.

For example:

```
<html>
        <script language="C#" runat=server>

                private void Page_Load(Object sender, EventArgs e) {

                        MyRepeater.Datasource = { "One", "Two", "Three" };
                        MyRepeater.DataBind();
                }

                private void ChangeBtn_Click(Object sender, EventArgs e) {

                        HtmlGenericControl x = MyRepeater.Items[0].FindControl("Message");

                        if (x != null) {
                                x.InnerHtml = "I just changed this item!!!";
                        }
                }

        </script>

        <body>
                <form action="Repeater.aspx" method="Post" runat=server>

                        <asp:repeater id="MyRepeater" runat=server>
                                <template name="itemtemplate">
                                        <span id="Message" runat=server>
                                                <%# Container.DataItem %>
                                        </span>
                                </template>
                        </asp:repeater>

                        <input type=submit OnServerClick=" ChangeBtn_Click" runat=server>

                </form>
        </body>
</html>
```

Note that an instance variable to all controls defined within the root-naming container (which is the Page instance) is automatically code-generated into the derived page class for convenience. For example, the "MyRepeater" control above did not have to be accessed via a call to FindControl("MyRepeater"), instead if could be accessed just as "MyRepeater".

3.3 ViewState Management

State management is one of the most difficult concepts for web developers (both new and old) to master. Programmers have often learned the hard way that storing state on the server has many problems (timeouts, huge memory allocations, webfarms, etc). The other alternative – storing state on the client – requires many tricks/hacks to get right (serialization of datatypes, URL encoding or data, identification of values, etc).

XSP's core HTTPRuntime infrastructure provides a rich collection of server-only state management services that enable web developers to store state information at either an application or user session level (see other specs for details). The WebForms Page Framework augments these options by enabling page and control developers to also persist "viewstate" information (state associated with a particular view of an HTML page).

Page/Control developers typically persist viewstate information using the *Control.ViewState* dictionary. This ViewStateBag instance automatically maintains a localized dictionary view of viewstate data – meaning two controls on the same page could persist out a value with a common key without colliding (the page framework uses a control's UniqueID to avoid conflicts).

Page/Control developers can read from the *ViewState* collection at anytime during a page request (although developers cannot write values into the collection during or after the "Render" portion of page execution).

```
public class Table : Control {

        // Expose a public "BGColor" property on the table control.  Store the value in viewstate
        // to ensure that the value is automatically round-tripped back to the server on postback.

        public String BGColor {
                get {
                        return ViewState["BGColor"];
                }
                set {
                        ViewState["BGColor"] = value;
                }
        }
}
```

At render time, the collective viewstate of all server controls on a page is output by server-side form controls into a specially named hidden form variable (__VIEWSTATE) on the generated html page:

```
<form>
        <input type=hidden name="__VIEWSTATE" value="<a><s>BgColor:red</s></a>">
        <input type=hidden name="__VIEWSTATEMAC" value="434343433433">

        <table bgcolor="red"> </table>
</form>
```

An additional hidden form variable (__VIEWSTATEMAC) is used to enable WebForms to execute a message authentication check on page post-back to ensure that the original viewstate contents remain unmodified between requests (note: this ensures that pages can't be spoofed by hackers). If the MAC of the incoming viewstate data is legal, the WebForms Page Framework automatically populates the *ViewState* dictionary with its previously saved state data immediately prior to the Init method on being called on each server control instance.

3.4 PostBack Data Services

The viewstate services described in the previous section enable each server control to cough up enough state information at the end of each request to enable it to restore itself to its last used condition on the next request (note: this might sound extremely expensive, but in reality it typically entails saving only a minimum amount of data). The end result is a programming paradigm that provides page developers with the perception that they are programming against a page that remains consistent on the server across multiple client round-trips.

When a post back occurs, it is important that any data modified by the user on the client is mapped back to the appropriate server control and reflected to the web developer on the server. This is handled within the WebForms Page Framework by the *IPostBackDataHandler* interface.

When a postback occurs, the Page Framework examines the posted contents looking for all values whose name matches the UniqueID of a server control that implements the IPostBackDataHandler interface. If a match is found, the framework calls the IPostBackDataHandler::LoadPostData method on the control, and supplies it with a HTTPValueCollection containing the appropriate post values. The control then consumes the values, compares them to its current state (set either from declarative initialization values or viewstate), and returns a boolean value indicating whether it has changed as a result of the postback. The page framework accumulates a list of all changed controls, and after Load()completes, uses the list to invoke the IPostBackDataHandler:: RaisePostDataChangedEvent method on all of those controls that have changed. These controls can then optionally fire an appropriate "change" event to any event listeners:

```
public class InputBox : Control, IPostBackDataHandler {

    private EventHandler        changeHandler;
    private String              text;

    public String Text {
        get { return ViewState["text"] }
        set { ViewState["text"] = value; }
    }

    // Register delegate to external "OnChange" event listener
    public void AddOnChange(EventHandler value)  {
        changeHandler = value;
    }

    // Return true if posted value does not match value on last page request
    public boolean LoadPostData(HTTPValueCollection values) {
        text = values[this.UniqueID];
        return (text == ViewState["previoustext"]));
    }

    // Raise appropriate change event to any "OnChange" listeners
    public void RaisePostDataChangedEvent() {
        if (changeHandler != null) {
            changeHandler(new EventArgs());
        }
    }
}
```

```
<html>
	<script language="VB" runat=server>

		Sub Name_Change(ByVal Source as Object, ByVal E as EventArgs)
			Message.Text = Name.Text & " is " & Age.Text & " years old!"
		End Sub

	</script>

	<body>
		<form action="Postback.aspx" method="POST" runat=server>

		  Name: <input id=Name type=text OnServerChange=Name_Change runat=server>
		  Age: <input id=Age type=text OnServerChange=InputBox_Change runat=server>

		  <input type=submit value="Click Me" runat=server> <br><br>

		  <span id=Message runat=server/>

		</form>
	</body>
</html>
```

Note: The data postback process is deliberately split over two methods (LoadPostData and RaisePostDataChangedEvent). This is to ensure that developers are always working against a consistent model of the page when the "OnChange" event firings occur. If the controls instead fired the "OnChange" events from within LoadPostData, it would be possible for a page developer responding to an "OnChange" event from one inputbox to use an old value pulled from another inputbox whose LoadPostData method had not yet been called.

3.5 PostBack Event Services

Pages within WebForms can be activated either as a result of a navigation request to the page (for example: following a link from another page or site), or alternatively as a result of a "postback" from a previously rendered instance of the same page on the client (for example: a registration form with multiple text fields that need to be submitted back and validated).

WebForms provides a built-in event infrastructure that enables controls to capture and process postback form submits from a client. This enables control developers to better encapsulate functionality (for example: a calendar control could automatically handle the "next month" and "previous month" button events from the client without requiring user-code to execute), and enables control consumers to more intelligently interact with them (for example: a button control could capture a postback and then expose a "OnServerClick" event that they raised on the server when a client user clicked it).

Controls indicate that they are interested in capturing postback events by implementing the IPostBackEventHandler interface. Postback event notifications – indicated by a call to the IPostBackEventHandler::RaisePostBackEvent method – can then be routed to the control in response to either a postback from a standard html form submit (ie: from something like a submit or image button), or as a result of a custom javascript method that initiates a submit being executed on the client.

Standard Form Submit

Html supports two intrinsic tag controls -- <input type=submit> and <input type=image> -- that when clicked will cause their containing form to be submitted to the server. Note that the name/value pair of one of these html elements is only included in the posted contents if it was the actual control initiating the submit – otherwise its value is excluded. For example:

```
<form action="Foo.aspx" method="GET">
        Name: <input type="text" name="Name">
        <input type=submit name="Button1">   |   <input type=submit name=Button2>
</form>
```

If "Button1" is clicked on the above form, the querystring:

```
        Foo.aspx?name=scott&Button1
```

will get posted. If "Button2" is clicked on the above form, then the querystring:

```
        Foo.aspx?name=scott&Button2
```

will get submitted.

WebForm Controls can easily render these three basic html intrinsic types - setting each ones "name" property to that of its "UniqueId" -- to capture postbacks on the server.

```
public class MyButton : Control, IPostBackEventHandler {

        private EventHandler    clickHandler;

        // Register delegate to external "OnServerClick" event listener
        public void AddOnServerClick(EventHandler value) {
                clickHandler = (EventHandler) Delegate.Combine(clickHandler, value);
        }

        // Remove delegate to external "OnServerClick" event listener
        public void RemoveOnServerClick(EventHandler value) {
                clickHandler = (EventHandler) Delegate.Remove(clickHandler, value);
        }

        protected void OnServerClick(EventArgs e) {
                if (clickHandler != null) {
                        clickHandler(this, EventArgs.Empty);
                }
        }

        // Process and raise appropriate incoming events
        public void RaisePostBackEvent(String eventArgument) {
                OnServerClick(EventArgs.Empty);
        }

        // Generate html button that will cause a postback to the control
        public override void Render(HTMLTextWriter output) {
                output.Write("<input type=submit name=" + this.UniqueId + ">");
        }
}
```

When a postback on the page occurs, the page framework will iterate over every posted value in the appropriate form/querystring collection, searching for a control in the hierarchy tree that implements the IPostBackEventHandler interface and whose UniqueID matches the value key. If found, the page framework will invoke its RaisePostBackEvent method (passing null as the argument value). The server control is then free to do anything it likes and/or raise appropriate control events (for example: "OnServerClick", "OnMoveNext", "OnCancel", etc) to external listeners.

Custom Script Event

Unfortunately not all html controls are capable of generating form submits back to the server. To enable UI widgets other than standard buttons and image buttons to also participate in the postback framework, WebForms supports a custom client-javascript eventing architecture that can optionally also be used to initiate form postbacks to the server.

This is accomplished using two hidden form fields and a small script function that are rendered on demand (by the html form control) to the client. The hidden form fields indicate which server control should be posted to, and optionally specify an argument to be passed. The small javascript function is used to set the hidden form field contents and initiate the actual form submit to the server.

For example, the below html demonstrates some sample output from a page with two "linkbutton" server controls that support postback events from html anchor style output:

```
<script language="JavaScript">
        function __doPostBack(eventTarget, eventArgument) {
                var theForm = document.HtmlForm0;
                theForm.__EVENTTARGET = eventTarget;
                theForm.__EVENTARGUMENT = eventArgument;
                theForm.submit();
        }
</script>

<form name="Form1">
        <input type="hidden" name="__EVENTTARGET" value="">
        <input type="hidden" name="__EVENTARGUMENT" value="">

        Click below links to cause postbacks:

        <a href="javascript:__doPostBack('LinkButton1', 'Click')"> LinkButton1 </a>
        <a href="javascript:__doPostBack('LinkButton2, 'Click')"> LinkButton2 </a>
</form>
```

When a postback on the page occurs, the page framework will examine the control tree hierarchy for a server control whose "UniqueId" property matches that specified in the "__EVENTTARGET" hidden field. The page framework will then query the target control for the IPostBackEventHandler interface and – if found – invokes the IPostBackEventHandler::RaisePostBackEvent method – passing the eventArgument field as an argument. For example:

```
public class MyLinkButton : Control, IPostBackEventHandler {

        private EventHandler    clickHandler;

        // Register delegate to external "OnServerClick" event listener, and then notify
        // the page instance that a client-side postback script block needs to be generated
        public void AddOnServerClick(EventHandler value) {
                clickHandler = (EventHandler) Delegate.Combine(clickHandler, value);
                Page.RegisterPostBackScript();
        }

        // Process and raise appropriate incoming events
        public void RaisePostBackEvent(String eventArgument) {
                if (clickHandler != null)
                        clickHandler(this, EventArgs.Empty);
        }

        // Generate html button and use appropriate helper method to get call back script
        public override void Render(HTMLTextWriter output) {
                output.Write("<a href="javascript:" + Page.GetPostBackEventReference(this)" + ">");
                output.Write("          " + this.UniqueId + " \n");
                output.Write("</a>");
        }
}
```

Note that the client-side javascript function is not generated by default during page rendering. Instead, a control must explicitly indicate that it wants this to be rendered by calling the Page.RegisterPostBackScript() method. They can then utilize the Page.GetPostBackEventReference helper methods to generate calls to the function (with appropriate parameters) at runtime.

3.6 DataBinding

WebForms supports a hierarchical DataBinding model that allows page developers to declaratively associate 1-way DataBinding expressions between data sources and server control properties. WebForms allows any server control property to be databound (the property is specified declaratively as an attribute name on the server control – see section 2.7 of this spec for full details).

DataBinding Expression Scope

DataBinding expressions are written in the default language of the page (declared via the "Page" directive) and can evaluate against any public field or property either on the page itself, or on their immediate naming container (by referencing the strongly typed "Container" instance pushed into the namespace of each binding expression).

DataBinding Expression Evaluation

DataBinding expressions are evaluated at runtime in response to a developer explicitly invoking the "Control.DataBind" method on a control in the tree hierarchy. This call causes a recursive tree walk from that control on down in the tree– causing the "OnDataBind" event to be raised on each server control in the hierarchy and DataBinding expressions on the control to be evaluated accordingly.

DataBinding Example:

```
<html>
        <script language="C#" runat=server>

                public void Page_Load(Object sender, EventArgs e) {

                        ArrayList myList = new ArrayList();

                        myList.Add("DataItem One");
                        myList.Add("DataItem Two");
                        myList.Add("DataItem Three");

                        MyRepeater.DataSource = myList;
                        MyRepeater.DataBind();
                }

        </script>

        <body>
                <asp:repeater id="MyRepeater" runat=server>

                        <template name="itemtemplate">

                                Here is the value in a text box:

                                <input type=text value="<%#Container.DataItem%>" runat=server>

                        </template>
                </asp:repeater>
        </body>
</html>
```

3.7 Template UI

One of the limitations of traditional forms frameworks is the lack of flexibility that *control consumers* have to customize instances of controls on a form/page. For example, a developer using a grid control in VB6 might be able to customize the background color and/or width/height. However, they would not be able to richly customize what each row of the grid contained (for example: substituting a calendar control in each row to format a date pulled from a database). Instead, they'd either have to write their own control from scratch or find another vendor to write it for them.

In a UI world as rich as HTML, where hierarchical containment (like tables and any kind of repetition) is standard and pre-canned pages are blasé, it is absolutely critical that we have a rich control customization story (lest all of our pages end up looking like boring VB Ruby Forms - instead of rich web apps like Sportszone, Yahoo, Amazon, etc).

The WebForms Template Customization Architecture provides such a customization infrastructure, and enables *control developers* to build controls that are as "lookless" as possible - allowing them to abstract control behavior away from their actual appearance. *Control consumers* can take advantage of this functionality to provide substitutable template parameters that enable rich customization of controls - allowing them to precisely control their look/feel and empowering them to enable scenarios that the original control developers might not have even realized possible.

ITemplate Instances

Control developers expose template customization support within their controls using get/set properties of type ITemplate (syntax details on how this can be done declaratively can be found in section 2.6 of the spec). Objects of type ITemplate provide a factory definition for customizing/ creating a control instance dynamically (by invoking the "ITemplate.Initialize" method).

For example, the sample below demonstrates how a repeater control could have a custom "ItemTemplate" declaratively specified within a page:

```
<html>
        <script language="C#" runat=server>

                public void Page_Load(Object sender, EventArgs e) {

                        ArrayList myList = new ArrayList();

                        myList.Add("DataItem One");
                        myList.Add("DataItem Two");
                        myList.Add("DataItem Three");

                        MyRepeater.DataSource = myList;
                        MyRepeater.DataBind();
                }

        </script>

        <body>
                <asp:repeater id="MyRepeater" runat=server>
                        <template name="itemtemplate">
                                Here is the value in a text box:
                                <input type=text value="<%#Container.DataItem%>" runat=server>
                        </template>
                </asp:repeater>
        </body>
</html>
```

The repeater control could then be implemented as follows:

```
public class RepeaterItem : Control, INamingContainer {

        public Object           DataItem;
        public int              Index;

        public RepeaterItem(Object dataItem, int index) {
                this.DataItem = dataItem;
                this.Index = index;
        }
}

public class Repeater : Control, INamingContainer {

        private ITemplate       itemTemplate;
        private ICollection     datasource;

        // Customizable template for each Item within the repeater

        public ITemplate ItemTemplate {
                get { return itemTemplate; }
                set { itemTemplate = value; }
        }

        // Datasource property to use to dynamically create rows within list

        public ICollection DataSource {
                get { return datasource; }
                set { datasource = value; }
        }

        // Framework method to dynamically create and initialize item rows in response
        // to a DataBind() call.

        protected override void OnDataBind(EventArgs e) {

                IEnumerator iterator = DataSource.GetEnumerator();
                int index = 0;

                while (iterator.GetNext()) {

                        // Construct a new RepeaterItem control instance - settings its dataitem
                        // and index properties dynamically using appropriate values

                        RepeaterItem item = new RepeaterItem(iterator.GetObject(), index);

                        // Customize the item instance using the declarative template

                        ItemTemplate.Initialize(item);

                        // Add the control instance into the hierarchy tree

                        this.Controls.Add(item);
                }
        }
}
```

4.0 User Controls

WebForm's server control architecture provides a powerful way for developers to encapsulate and reuse functionality across multiple pages. Unlike existing technologies today like server-side includes, server controls provide complete encapsulation of code and content. There are no variable/language conflicts between the control author and control consumer, the control author is given the flexibility to expose only those properties/methods/events that they wish to make public to external consumers, and the page control framework automatically handles form field naming conflicts (since each control is provided a "UniqueID" which is hierarchical and guaranteed to be unique across all controls on a page). In short, we think they are very, very cool and want to make sure that developers populate the world with many different variations of them... :-)

WebForms enables third-party developers to create a new server control by either compiling a new COM+ Class that extends the "Control" base class (examples of this have been seen in the previous section of the spec), or by authoring a "user control". User controls provide an easy, content-centric, way to partition and compose common functionality across pages - and help "humanize" the control writing process. Our goal is to ensure that anyone who can write an page can turn around and write a user control. Indeed, developers should be able to just copy/paste existing page code/content from an existing file into a user control, hit save, and then be able to re-use it as a control from another page.

Building a user control

Developers create user controls by authoring a text file whose format is similar to a regular page (although some attributes - like SessionState, OutputCache, etc are not valid on a user control).

```
<%@ Control Description="A Simple User Control" %>

<script language="C#" runat=server>

    public void Page_Load(Object sender, EventArgs e) {

        if (!IsPostBack) {
            MessageBox.InnerHtml = "Here is the default message...";
        }
    }

    public String Message {
        get {
            return MessageBox.InnerHtml;
        }
        set {
            MessageBox.InnerHtml = value;
        }
    }
</script>

<table>
    <tr><td>Here is my simple control...</td></tr>
    <tr><td> <span id="MessageBox" runat=server/> </td></tr>
</table>
```

The text file can be saved with any file name or extension (it is perfectly legal to call it ".aspx"). Developers may optionally provide the file with a ".ascx" file extension if they do not want browser clients to be able to download/access it directly (note that XSP script-maps this extension to automatically block direct access to these files).

Using a User Control

Page developers reference and use user controls by first importing them into a page via an Reference directive that specifies: 1) the source location of the control, 2) a friendly tag name that can be used to declare an instance of it on the page.

For Example:

```
<%@ Page Description="A Sample Page Consuming a User Control" %>
<%@ Register TagPrefix="Scottgu" TagName="FooBar" Src="MySimpleControl.aspc" %>

<html>
        <script language="VB" runat=server>

                Sub Btn_Click(ByVal Source as Object, ByVal E as EventArgs)
                        MySimpleControl.Message = "The button was just clicked..."
                End Sub

        </script>

        <body>
                <form runat=server>

                        <ScottGu:FooBar id="MySimpleControl" Message="Hello There" runat=server/>

                        <input type=Submit Value="Click Me!" OnServerClick="Btn_Click" runat=server>

                </form>
        </body>
</html>
```

5. Code Behind

WebForms supports two page development modes: 1) where page logic code is written within <script runat=server> blocks within a ".ASPX" file and dynamically compiled the first time the page is requested on the server, 2) where page logic code is written within an external COM+ class that is compiled prior to deployment on a server and linked "behind" the .ASPX file at runtime.

Developers writing external classes must ensure that they extend the Page base class (or from UserControl if developers are building a code-behind user control). At runtime WebForms will dynamically extend the compiled class using the ".ASPX" file and can automatically take care of wiring up references to contained server controls whose name and type signatures match a member reference declared on the compiled class (provided the member reference is declared as either "public" or "protected"). XSP will also automatically take care of wiring up appropriate delegate objects for declarative event bindings.

For example, the below class and ".ASPX" file demonstrate how a developer could take advantage of "code behind" page development to cleanly separate their logic and presentation code.

MyPage.cs Code-Behind Class:

```
using System.Web;
using System.Web.UI;
using System.Web.UI.HtmlControls;

public class MyPage : Page {

        protected HtmlControl                   Message     =     null;
        protected HtmlInputButton               MyBtn       =     null;

        public void Page_Load() {
                If (!IsPostBack) {
                        Message.InnerHtml = "This is the first time the page has been hit by you!";
                }
        }

        protected void MyBtn_Click(Object source, EventArgs e) {
                Message.InnerHtml = "Wow -- You just pressed the button!";
        }
}
```

Declarative ".ASPX" File:

```
<%@ Page Description="This is a sample code-behind page" Inherits="MyPage" Src="MyPage.cs"%>

<html>
        <body>
                <form runat=server>

                        <h1> <span id="Message" runat=server/> </h1>

                        <input type=submit value="Push" OnServerClick="MyBtn_Click" runat=server>

                </form>
        </body>
</html>
```

6. Trace Functionality

Building complex web applications can be a challenging task. Inadequate debugging tools and/or weak instrumentation data often compound the problem. Our goal with XSP is to improve this by leveraging the COM+ Debugging infrastructure to provide powerful *cross-language* debugging support that can be used both *locally* and *remotely* from a web server. We will also be augmenting this within the page framework by providing a powerful trace mode capability that enables both controls and page developers to append instrumentation messages throughout their code. These instrumentation messages can be used to simplify the process of identifying just what exactly occurs during a given web request, and help track down resulting problems.

Writing to the Trace Log

Page and control developers can write to the trace log using the "Trace" property exposed on the Control base class. Developers can organize each output message using a provided "category" string.

Sample Control Using Tracing:

```
public class MyButton : Control {

        public override void Render(HTMLTextWriter output) {

                Trace.Write("Render", "Button class is about to render....");

                output.Write("<button>Simple Button</button>");

                Trace.Write("Render", "Button class has just finished rendering....");
        }
}
```

Sample Page Using Tracing:

```
<%@ Page Language="VB" Trace=True %>

<html>
        <script runat=server>

                Sub Page_Load(ByVal Sender as Object, ByVal E as EventArgs)
                        Trace.Write("Scott's Category", "Hey buddy - we're at Load()!!!")
                End Sub

        </script>

        <body>
                <form runat=server>
                        <span id="Message" runat=server/>
                </form>
        </body>
</html>
```

Output Trace Messages

By default tracing is not enabled on a page - which means that "Trace.IsEnabled" will return false and that no messages written to the trace log will be output during a web request.

Developers can optionally enable tracing by setting the "Trace" attribute on the "Page" directive to "true". For example:

```
<%@ Page Language="VB" Trace=True %>
```

In addition to causing the "Trace.IsEnabled" property to return "true", this will cause the Page class to automatically output an HTML table at the conclusion of page rendering that details all of the assorted category trace logs. For example:

```
<html>
        <body>
                <form runat=server>
                        <span id="Message" runat=server/>
                </form>
        </body>
</html>

<table>
        <tr>
                <td colspan=2><h1>Trace Log</h1></td>
        </tr>

        <tr>
                <td><h3>Scott's Category</h3></td>
                <td> Hey buddy - we're at Load()!!! </td>
        </tr>

        <tr>
                <td><h3>Render Category</h3></td>
                <td>
                        Button class is about to render.... <br>
                        Button class has just finished rendering....
                </td>
        </tr>
</table>
```

Developers can optionally toggle the order/presentation of trace messages using the "tracemode" attribute exposed on the page directive. A value of "SortByTime" will cause the trace messages to be output in the order in which they were written. A value of "SortByCategory" will cause the trace messages to be output alphabetically by category. A value of "ShowInline" will cause the messages to be emitted immediately after Trace.Write is called.

Note that in addition to outputting user-provided trace content, the page class will also automatically include additional performance data, tree-structure information, and state management content.

7. WebForms Page Parsing, Transformation and Compilation

XSP provides a low-level request/response API that enables developers to use COM+ Runtime classes to service incoming HTTP requests. Developers accomplish this by authoring classes that support the System.Web.IhttpHandler interface and implement the ProcessRequest() method. Each incoming HTTP request received by an XSP Application Server is ultimately processed by a specific instance of an IHTTPHandler class.

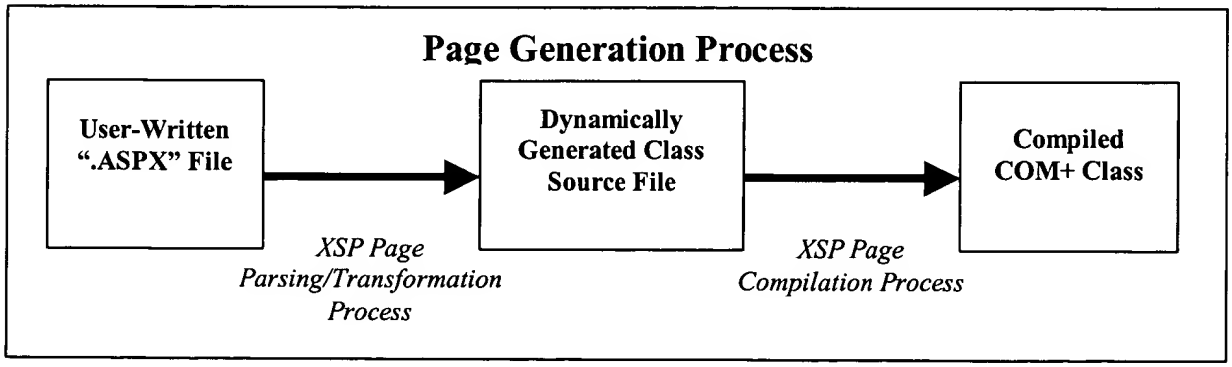
IHTTPHandler Factories provide a pluggable architecture that handles the actual resolution of URL requests to IHTTPHandler instances. This resolution is facilitated using application configuration settings that can map an incoming URL's file-extension + HTTP verb to an IHTTPHandlerFactory class ultimately responsible for surfacing an appropriate IHTTPHandler instance. Full details about the IHTTPHandlerFactory architecture can be found in the HTTPHandlerFactory specification.

PageFactory Class

The System.Web.UI.PageFactory class provides an IHTTPHandlerFactory implementation that handles the instantiation and configuration of all WebForm Pages. By default, all ".aspx" URLs are configured for processing by the PageFactory class.

When a request for a particular ".aspx" resource is first received by a PageFactory class, it searches the filesystem for the appropriate path-translated text-file. If the physical file exists, the PageFactory opens and reads it into memory. If the file cannot be found, the PageFactory returns an appropriate "file not found" error message.

Once opened and read into memory, the PageFactory class parses the file contents to build-up a data model of the page (lists of script blocks, directives, static text regions, hierarchy of server controls, server control properties, etc). This data model is then used to generate a source listing of a new COM+ class that extends the System.Web.UI.Page base class. This source listing is then dynamically compiled into COM+ IL and later JIT'd (Just-In-Time compiled) by COM+ into platform native instructions (x86, Alpha, WinCE, etc). Note that both the compilation and PE persistence of the resulting class are all done in-memory - at no point does anything need to be written to disk.



The end product of a page's transformation->compilation process is a reference to a "System.Type" object that provides meta-data and reflection access to the COM+ class, as well as APIs for dynamically creating new instances of it. The PageFactory will cache this System.Type object as long as the originating text file remains unchanged. If modifications are made, the cached "System.Type" reference will simply be thrown away and a new COM+ class built.

The PageFactory creates a new instance of a page class to service each incoming web request (our assumption is that 99.9% of developers – including most of those who work at Microsoft – cannot write multi-threaded code). It accomplishes this simply by calling the "CreateInstance()" method on the appropriate "System.Type" reference.

Page Parsing/Transformation

The WebForms Page Parser/Transformation Engine is responsible for taking a declarative ".aspx" file as input and dynamically generating a new COM+ class source file that – when compiled and run – faithfully represents the developer's desired intentions. For example, the "birthday voting" page:

```
<%@ Page Language="VB" Description="Simply page that prompts user for their birthday" %>

<html>
        <script runat=server>

                Dim PollVoter as New BirthdayVoterBizObject

                Sub Page_Load(Sender as Object, E as EventArgs)
                        BirthdayVoter.Server = "\\votingcentral"
                        BirthdayVoter.Category = "personal_birthday"
                        BirthdayVoter.Timeout = 20
                End Sub

                Sub MyCal_DaySelect (ByVal Source as Object, ByVal Details as DaySelectEvent)
                        BirthdayVoter.SubmitBirthday(Details.CurrentDay)
                End Sub

        </script>

        <body>
                <wfc:adrotator file="myAds.xml" category="birthday" runat="server"/> <br>

                <h1> Please Enter your Birthday: </h1>

                <form method="post" runat=server>

                        <wfc:calendar id="myCal" runat="server" onDaySelect="MyCal_DaySelect">

                                <template name="title">
                                        <i>Birthday Picker</i>
                                </template>

                                <template name="daytemplate">
                                        <table>
                                                <td>
                                                        <font size=2> <%= DataContainer.Day %> </font>
                                                </td>
                                        </table>
                                </template>

                        </wfc:calendar>

                </form>
        </body>
</html>
```

Would cause the below source file to be generated:

```
' XSP Generated Code ------------------------------------------------------------------------

Inherits Page

Dim __Page_TextLiteral1 as New System.Web.UI.Controls.TextLiteral
Dim __Page_AdRotator1  as New Microsoft.Wfc.WebForms.AdRotator
Dim __Page_TextLiteral2 as New System.Web.UI.Controls.TextLiteral
Dim __Page_FormControl1 as New System.Web.UI.Controls.Form
Dim myCal as New Microsoft.Wfc.WebForms.Calendar
Dim __Page__MyCal_TemplateTitleBuilder as New ControlCollectionBuilder
Dim __Page__MyCal_TemplateDayBuilder as New ControlCollectionBuilder
Dim __Page_TextLiteral3 as New System.Web.UI.Controls.TextLiteral
Dim __Page_TextLiteral4 as New System.Web.UI.Controls.TextLiteral
Dim __Page_CodeBlock1 as New System.Web.UI.Controls.CodeBlock
Dim __Page_TextLiteral5 as New System.Web.UI.Controls.TextLiteral
Dim __Page_TextLiteral6 as New System.Web.UI.Controls.TextLiteral

' User Written Code ---------------------------------------------------------------------------

Dim PollVoter as New BirthdayVoterBizObject

Sub Page_Load(Sender as Object, E as EventArgs)
        BirthdayVoter.Server = "\\votingcentral"
        BirthdayVoter.Category = "personal_birthday"
        BirthdayVoter.Timeout = 20
End Sub

Sub MyCal_DaySelect (ByVal Source as Object, ByVal Details as DaySelectEvent)
        BirthdayVoter.SubmitBirthday(Details.CurrentDay)
End Sub

' XSP Generated Constructor -------------------------------------------------------------------

Sub New()
        __Page_TextLiteral1.Value = "<html>\r\n      <body>\r\n"
        Me.Controls.Add(__Page_TextLiteral1)

        __Page_AdRotator1.File = "myAds.xml"
        Me.Controls.Add(__Page_AdRotator1)

        __Page_TextLiteral2.Value = " <br>\r\n      <h1> Please Enter Your Birthday: </h1>\r\n"
        Me.Controls.Add(__Page_TextLiteral2)

        __Page__FormControl1.Method = "post"
        Me.Controls.Add(__Page_FormControl1)

        MyCal.AddOnDaySelect(New DaySelectHandler(AddressOf MyCal_DaySelect))
        Set __Page__MyCal_TemplateTitle.Build = AddressOf __Page_MyCal_TitleTemplateBuild
        Set MyCal.TitleTemplate = __Page__MyCal_TemplateTitle
        Set __Page__MyCal_TemplateDayBuilder.Build = AddressOf __Page_MyCal_DayTemplateBuild
        Set MyCal.DayTemplate = __Page__MyCal_TemplateDayBuilder
        __Page_FormControl1.Controls.Add(MyCal)

        __Page_TextLiteral6.Value = "    </form>\r\n      </body>\r\n</html>"
        Me.Controls.Add(__Page_TextLiteral6)
End Sub
```

```
Sub __Page_MyCal_TitleTemplateBuild(MyControlCollection as ControlCollection)

        __Page_TextLiteral3.Value = "<i>Birthday Picker</i>\r\n"
        MyControlCollection.Add(__Page_TextLiteral1)

End Sub

Sub __Page_MyCal_DayTemplateBuild(MyControlCollection as ControlCollection)

        __Page_TextLiteral4.Value = "<table>\r\n    <td>\r\n   <font size=2>\r\n"
        MyControlCollection.Add(__Page_TextLiteral4)

        __Page_CodeBlock1.RenderDelegate = AddressOf (__Page_CodeBlock1Render)
        MyControlCollection.Add(__Page_CodeBlock1)

        __Page_TextLiteral5.Value = "</font>\r\n      </td>\r\n   </table>\r\n"
        MyControlCollection.Add(__Page_TextLiteral5)

End Sub

Sub __Page_CodeBlock1Render(DataContainer as Object)

        Response.Write(DataContainer.CurrentDate)

Enb Sub
```

It is not necessary for the page developer (or even the reader of this spec) to understand what the above source-file does or how it works. However, for the geeks (and advanced server control developers) out there, the below implementation notes might provide some interest:

- By default the generated source-file subclasses the "System.Web.UI.Page" base class. Developers can optionally specify an alternative class using the "Inherits" attribute provided on the "Page" directive.

- All code written within a code declaration block -- <script runat=server> </script>) - is conceptually treated as page member declarations (variables, properties, methods) and is directly inserted into the source file of the generated file.

- Everything not written within either a code declaration block -- <script runat=server> </script> -- or a directive -- <%@ Directive attribute=value %> -- is considered part of the "output canvas" of a page. This "output canvas" is composed from one or more server control elements arranged using a hierarchical, tree-based, containership model.

- All server control tags that contain a unique "id" value and are not contained within a parent <template:templatename> element are automatically declared as strongly typed member variables of the generated page class (for example: the "myCal" server control in the sample above).

- A server control tag's attribute values will by default be converted to typed property sets on the resulting server control class instance. Attribute names that correspond to server control events (for example: the wfc:calendar's OnDayClick attribute/event in the sample above) will cause appropriate "AddOnEventName" delegate wiring code to be generated to the page method with the same name as the server control tag's attribute value.

- A server control can be nested/contained within another control (for example: in the above sample the "wfc:calendar" control is nested as a child element of the "form" control). The dynamically generated page class always serves as the "root control" within the composition hierarchy.

- A server control can optionally utilize "inline template" to enable advanced customization of a server control. Support for these templates is provided by a server control developer by exposing an ITemplate typed property. When the WebForms Parser/Transformation Engine encounters a <template:templatename> tag within an page file whose templatename matches one of these typed properties, it assigns it an appropriate builder object that holds a private delegate reference to the template's "build method" on the page class. Server Control developers can utilize this builder object to lazily create zero or more instances of the template tree at runtime (for example: the "wfc:calendar" control might create up to 31 separate instances of the "daytemplate").

- Static literal content (text that does not describe either a server control or a <% %> code block) is collapsed into "LiteralText" controls that represent ranges of text. These controls are inserted into the page's hierarchy tree (their implementation of the "Control.Render" method simply outputs the appropriate text range string). The text range string for each LiteralText control is allocated statically (once per-class - not per instance) to cut-down on memory allocations.

- WebForms Code Render Blocks (<% %> and <%= %>) are injected as custom "render methods" on the resulting page class (so that they can access variables/properties/methods declared within code declaration blocks (<script runat=server> </script>) and on the page's superclass. "CodeBlock" controls (which use a COM+ delegate pointer to invoke the appropriate render method) are then added to the server control hierarchy tree to maintain the inline code's position within the overall "output canvas".

- All member variables and methods dynamically produced by the WebForm Transformer are declared with a "__Page_" prefix. WebForms reserves the semantic ownership of this coding notation (in other words no one else should use this).

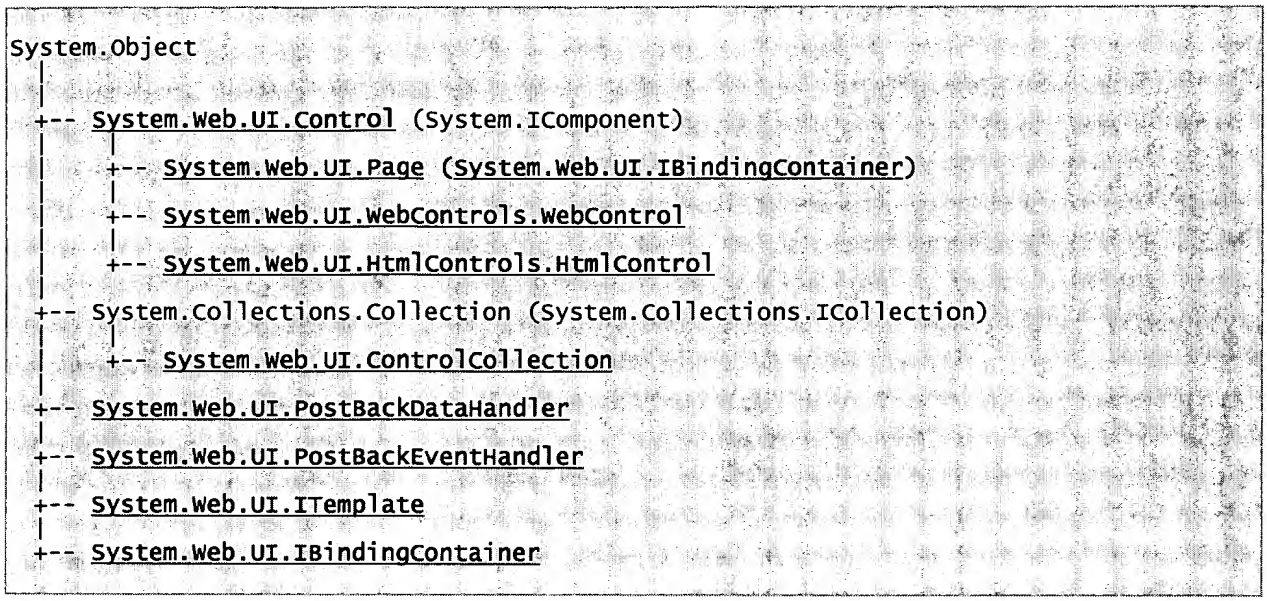
- Source code generated by the WebForms Page Parser/Transformer contains metadata markup (not shown in the example above) that enables a compiler/debugger to correctly resolve all code within the dynamically generated source file with its column/line position in the original ".aspx" file. This enables developers to set a breakpoint within a <script runat=server></script> block on the original file without ever having to see the dynamically generated version.

Page Class Execution

All operations necessary to setup, activate, and run a page are encapsulated within the dynamically compiled COM+ class. As a result, no additional configuration/parsing of files is necessary during page setup (the original ".aspx" file is never touched again), and a "runtime host environment" (ala an ASP script engine) is not required during page execution.

8. Class Definitions

The WebForms Page Framework is implemented within the System.Web.UI namespace:



The following sections of this specification describe each of the above classes/interfaces in detail.

8.1 System.Web.UI.Control

The Control class defines the methods, properties and events common to all server-controls within the WebForms Page framework.

```
namespace System.Web.UI
{
   public abstract class Control : IComponent
   {
      // Properties

      public String                              ID                  { get; set; }
      public Control                             Parent              { get; }
      public ControlCollection                   Controls            { get; }
      public Boolean                             MaintainState       { get; set; }
      public Control                             NamingContainer     { get; }
      public String                              UniqueID            { get; }
      public boolean                             Visible             { get; set; }

      protected HttpContext                      Context             { get; }
      public Page                                Page                { get; }
      protected ViewStateBag                     ViewState           { get; }

      // Methods

      public void                                DataBind();
      public Control                             FindControl(String controlID);
      protected virtual void                     CreateChildControls();
      public bool                                HasControls();

      protected virtual void                     LoadState(Object value);
      protected virtual Object                   SaveState();
      protected virtual void                     Render(HtmlTextWriter writer);
      public virtual void                        Dispose();

      // Events

      public event EventHandler                  Init;
      public event EventHandler                  Load;
      public event EventHandler                  PreRender;
      public event EventHandler                  Unload;

      public event EventHandler                  DataBind;
      public event BubbleCommandEventHandler     BubbleCommand;

      protected virtual void                     OnInit(EventArgs e);
      protected virtual void                     OnLoad(EventArgs e);
      protected virtual void                     OnPreRender(EventArgs e);
      protected virtual void                     OnUnload(EventArgs e);

      protected virtual void                     OnDataBind(EventArgs e);
      protected virtual void                     OnBubbleCommand(BubbleCommandEventArgs e);
   }
}
```

Properties

ID	
Prototype	public String ID { get; set; }
Description	Control name identifier

Parent	
Prototype	public Control Parent { get; set; }
Description	Reference to a Control's parent control in the UI hierarchy tree

Controls	
Prototype	public ControlCollection Controls { get; }
Description	Object Model Collection of a Control's children in the UI hierarchy tree

MaintainState	
Prototype	public bool MaintainState { get; set; }
Description	Indicates whether control should persist its viewstate (and the viewstate of its children) at the end of the current page request.

NamingContainer	
Prototype	public Control INamingContainer { get; set; }
Description	Reference to the current control's namingcontainer. This is defined as the first parent control up the hierarchy which implements the INamingContainer marker interface.

UniqueId	
Prototype	public String UniqueId { get; }
Description	Obtains unique – hierarchically qualified – string identifier for a control

Visible	
Prototype	public bool Visible { get; set; }
Description	Indicates whether control should be rendered

Context	
Prototype	protected HttpContext Context { get; }
Description	Reference to the HttpContext of the current web request.

ViewState	
Prototype	protected ViewStateBag ViewState { get; }
Description	Dictionary of ViewState information that enables control developers to save and restore the state of a control across multiple page requests.

Methods

DataBind	
Prototype	public void DataBind();
Description	Explicit method that causes DataBinding to occur on the invoked control – as well as recursively on all the controls within its children hierarchy.
Parameters	None

CreateChildControls()	
Prototype	protected virtual void CreateChildControls();
Description	Framework method that signals controls that utilize composition-based implementation that they should create their child controls in preparation for either postback or rendering.
Parameters	None

HasControls()	
Prototype	public bool HasControls();
Description	Framework perf optimization method that returns whether the control has any child controls. This method allows us to avoid having to call Control.Controls.Count (which would force us to create a new ControlCollection even if no children were present).
Parameters	None

FindControl	
Prototype	public Control FindControl(String controlID);
Description	Framework method that searches current naming container for a control whose ID value matches the controlID argument
Parameters	ControlID – ID of control to find

LoadState	
Prototype	public virtual void LoadState(Object state);
Description	Advanced framework method that enables controls to custom-restore view state information from a previous web request.
Parameters	State (Object) – Arbitrary object containing viewstate values returned from call to SaveState() on a previous web request.

SaveState	
Prototype	protected virtual Object SaveState();
Description	Advanced framework method that enables controls to custom-persist view state information for use on another web request.
Parameters	None

Render	
Prototype	protected virtual void Render(HtmlTextWriter output)
Description	Enables control to render appropriate content into the output stream being sent down to the client browser.
Parameters	*Output* – TextWriter (with built-in Html semantics) that can be used to render appropriate content

Dispose	
Prototype	public virtual void Dispose();
Description	Enables a control to perform final cleanup work.
Parameters	None

Events

OnInit	
Description	Event that signals controls that they should perform any initialization logic required to create and setup a control. Control authors cannot use ViewState information within this event (it is not populated yet). They should also avoid accessing other controls either in their child or parent hierarchy (these are not guaranteed to be created and ready for access yet).

OnLoad	
Description	Event that signals controls that they should perform any work that needs to occur on each page request. Control authors can use ViewState information within this event. They can also access other controls within the page.

OnPreRender	
Description	Event that signals controls that they should perform any pre-rendering steps necessary prior to saving view state and rendering the content.

OnUnload	
Description	Event that signals controls that they should perform final cleanup work before the control instance is torn-down.

OnDataBind	
Description	Event that signals controls that they should perform any DataBinding logic.

8.2 System.Web.UI.ControlCollection

The ControlCollection class provides an object-model style collection container that enables Controls to maintain lists of child controls.

```
namespace System.Web.UI
{
    public class ControlCollection : ICollection
    {
        // Properties

        public Control []          All                 { get; set; }
        public Control             this [int index]    { get; }
        public override int        Count               { get; }

        // Methods

        public void                Add(Control child);
        public int                 IndexOf(Control child);
        public IEnumerator         GetEnumerator()
        public void                Remove(Control child);
        public void                RemoveAt(int index);
        public void                Clear();
    }
}
```

Properties

All	
Prototype	public Control [] All { get; set; }
Description	Snapshot array of all child controls, ordered by index

this[int index]	
Prototype	public Control this[int index] { get; }
Description	Returns reference to ordinal-indexed control within the collection

Count	
Prototype	public int Count { get; }
Description	Number of child controls in collection

Methods

Add	
Prototype	public void Add(Control child);
Description	Adds specified control to collection
Parameters	*Child* – Control to add to collection

IndexOf	
Prototype	public int IndexOf(Control child);
Description	Returns ordinal index of control instance in collection. Returns -1 if instance is not currently a member of the collection.
Parameters	*Value* – Control to obtain ordinal index of

GetEnumerator	
Prototype	public IEnumerator GetEnumerator();
Description	Returns enumerator of all controls within collection
Parameters	None

Remove	
Prototype	public void Remove(Control value);
Description	Removes specified control instance from collection
Parameters	*Value* – Control reference to remove from collection

RemoveAt	
Prototype	public void RemoveAt(int index);
Description	Removes specified control from collection
Parameters	*Index* – Ordinal index of control to remove from collection

Clear	
Prototype	public void Clear();
Description	Removes all controls from collection

8.3 System.Web.UI.ViewStateBag

The ViewStateBag class is a helper class that is used to manage the state of properties. The class stores name/value pairs as string/object and tracks modification of properties after the Init() phase of execution is completed (only values modified after this period are actually persisted into viewstate). This class is the primary storage mechanism for all HtmlControls and WebControls.

```
namespace System.Web.UI
{
    public class ViewStateBag : ICollection
    {
        // Properties

        public Object              this [String key]        { get; set;}
        public Collection          Keys                     { get; }
        public Collection          Values                   { get; }
        public override int        Count                    { get; }

        // Methods

        public void                Add(String key, Object value);
        public void                Remove(String key);
        public void                Clear();
    }
}
```

Properties

this[String key]	
Prototype	public Object this[String key] { get; set; }
Description	Returns reference to state bag item

Keys	
Prototype	public Collection Keys { get; }
Description	Returns collection of all keys in state bag

Values	
Prototype	public Collection Values { get; }
Description	Returns collection of all values in state bag

Count	
Prototype	public int Count { get; }
Description	Number of state items in bag

Methods

Add	
Prototype	public void Add(String key, Object value);
Description	Adds specified state item into state bag
Parameters	*Key* – Key to state item in bag *Value* – Value to add to state bag

Remove	
Prototype	public void Remove(String key);
Description	Removes specified state item from bag
Parameters	*Key* – Key of item to remove

Clear	
Prototype	public void Clear();
Description	Removes all controls from collection

8.4 System.Web.UI.HtmlTextWriter

The HtmlTextWriter class provides rich formatting capabilities that Controls can leverage when rendering content to clients.

```
namespace System.Web.UI
{
    public class HtmlTextWriter : TextWriter
    {
        // Methods

        public void             Indent();
        public void             Unindent();

        public override void    Write(String s);
        public override void    Write(bool value);
        public override void    Write(char value);
        public override void    Write(char [] buffer);
        public override void    Write(char [] buffer, int index, int count);
        public override void    Write(double value);
        public override void    Write(float value);
        public override void    Write(int value);
        public override void    Write(long value);
        public override void    Write(Object value);
        public override void    Write(String format, Object arg0);
        public override void    Write(String format, Object arg0, Object arg1);
        public override void    Write(String format, Object arg0, Object arg1, Object arg2);
        public override void    Write(String format, Object [] arg);

        public override void    WriteLine();
        public override void    WriteLine(String s);
        public override void    WriteLine(bool value);
        public override void    WriteLine(char value);
        public override void    WriteLine(char [] buffer);
        public override void    WriteLine(char [] buffer, int index, int count);
        public override void    WriteLine(double value);
        public override void    WriteLine(float value);
        public override void    WriteLine(int value);
        public override void    WriteLine(long value);
        public override void    WriteLine(Object value);
        public override void    WriteLine(String format, Object arg0);
        public override void    WriteLine(String format, Object arg0, Object arg1);
        public override void    WriteLine(String format, Object arg0, Object arg1, Object arg2);
        public override void    WriteLine(String format, Object [] arg);

        public void             WriteAttribute(String name, String value);
        public void             WriteAttribute(String name, String value, EncodingTypeEnum type);
        public void             WriteBeginTag(String tagName);
        public void             WriteFullBeginTag(String tagName);
        public void             WriteEndTag(String tagName);
        public void             WriteEventHandler(Control control, String controlEvent);
    }
}
```

3.5 System.Web.UI.BubbleCommandEventArgs

The BubbleCommandEventArgs class serves as the EventArgs instance provided to all "bubbled" events in the WebForms framework. Listeners can use it to determine the actual Command that was bubbled, as well as an optional Argument detailing what happened.

```
namespace System.Web.UI
{
    public class BubbleCommandEventArgs : CancelEventArgs
    {
        // Properties

        public String    BubbleCommand      { get; }
        public Object    BubbleArgument     { get; }
    }
}
```

Properties

BubbleCommand	
Prototype	public String BubbleCommand { get; }
Description	Command description that has been bubbled up from a control. For example: "Sort", "Cancel", "Edit"
Default Value	null

BubbleArgument	
Prototype	public Object BubbleArgument { get; }
Description	Optional command argument that can be used by a listener to perform some type of additional action. For example, the BubbleArgument of a "Sort" BubbleCommand might be "Ascending" or "Decending".
Default Value	null

8.6 System.Web.UI.Page

The Page class defines the methods, properties and events common to all server-manipulated pages within the WebForms Page framework.

```
namespace System.Web.UI
{
    public class Page : Control, INamingContainer, IHttpHandler
    {
        // Properties

        public HttpRequest          Request         { get; }
        public HttpResponse         Response        { get; }
        public HttpSessionState     Session         { get; }
        public HttpApplicationState Application     { get; }
        public HttpServerUtility    Server          { get; }
        public TraceContext         Trace           { get; }

        public String               ErrorPage       { get; set; }
        public bool                 IsPostBack      { get; }
        public bool                 IsValid         { get; }
        public ValidatorCollection  Validators      { get; }

        // Methods

        public UserControl          LoadControl(String path);
        public ITemplate            LoadTemplate(String path);

        public void                 RegisterClientScriptBlock(String key, String script);
        public void                 RegisterHiddenField(string fieldName, string fieldValue);

        public void                 RegisterPostBackScript();
        public String               GetPostBackEventReference(Control control);
        public String               GetPostBackEventReference(Control control, string arg);

        public virtual void         HandleError(Exception e);
    }
}
```

Properties

Request	
Prototype	public HttpRequest Request { get; }
Description	HTTPRuntime provided request intrinsic that allows developers to access incoming HTTP request data

Response	
Prototype	public HttpResponse Response { get; }
Description	HTTPRuntime provided response intrinsic that allows developers to transmit HTTP response data to a client.

Application	
Prototype	public HttpApplicationState Application { get; }
Description	HTTPRuntime provided application intrinsic.

Session	
Prototype	public SessionState Session { get; }
Description	HTTPRuntime provided session intrinsic.

Context	
Prototype	public HttpContext Context { get; }
Description	HTTPRuntime provided context object that allows developers to gain access to additional pipeline-module exposed objects.

Server	
Prototype	public HttpServerUtility Server { get; }
Description	ASP Compatible "Server" Intrinsic

ErrorPage	
Prototype	public String ErrorPage { get; set; }
Description	ErrorPage to redirect to in the event of an unhandled page exception

IsPostBack	
Prototype	public bool IsPostBack { get; }
Description	Indicates whether the page is being loaded and accessed the first time, or in response to a client postback.

IsValid	
Prototype	public bool IsValid { get; }
Description	Indicates whether all validation logic on the page succeeded or failed. Please review the webforms validation spec for more details

Validators	
Prototype	public ValidationCollection Validators { get; }
Description	Collection of all validators on the page. Please review the webforms validation spec for more details.

Methods

LoadControl	
Prototype	public Control LoadControl(String path);
Description	Enables page/control developers to obtain a control instance from a declarative user control file.
Parameters	*Path* – Virtual Path to .aspx file or user control

LoadTemplate	
Prototype	public ITemplate LoadTemplate(String path);
Description	Enables page/control developers to obtain an ITemplate instance from an external file on disk.
Parameters	*Path* – Virtual Path to template file.

RegisterClientScriptBlock	
Prototype	public void RegisterClientScriptBlock(String key, String script);
Description	Enables controls to eliminate duplicate blocks of client-side script code being sent down to the client (any script with the same key value is considered a duplicate).
Parameters	*Key* – Unique key that identifies script (ie: "Wfc.Calendar_LinkHover") *Script* – Actual script content to write out with page content

RegisterHiddenField	
Prototype	public void RegisterHiddenField(string fieldName, string fieldValue);
Description	Enables controls to automatically register a hidden field on the form that will be emitted when the form control renders itself.
Parameters	*fieldName* – Unique name of hidden field to render *fieldValue* – Hidden form value to emit

RegisterPostBackScript	
Prototype	public void RegisterPostBackScript();
Description	Called by controls that require the __doPostBack javascript handler on the client. Can be called multiple times (by different controls). Only one instance of the __doPostBack script should be rendered.
Parameters	None

GetPostBackEventReference	
Prototype	public String GetPostBackEventReference(Control control);
Description	Enables controls to obtain client-side script function that will cause (when invoked) a server post-back to the form.
Parameters	*control* – Control that will handle the post-back on the server

GetPostBackEventReference	
Prototype	public String GetPostBackEventReference(Control control, string argument);
Description	Enables controls to obtain client-side script function that will cause (when invoked) a server post-back to the form.
Parameters	*control* – Control that will handle the post-back on the server *argument* – Parameter that will be passed to control on server

HandleError	
Prototype	public virtual void HandleError(Exception err);
Description	Enables page developers to catch and optionally handle runtime execution errors that occur during page execution
Parameters	*Err* – Exception information object detailing cause/location of error

8.7 System.Web.UI.IPostBackEventHandler

The IPostBackEventHandler interface defines the contract that Controls that wish to handle low-level postback events should implement.

```
namespace System.Web.UI
{
    public interface IPostBackEventHandler
    {
        // Post-Back Event Raising Method

        public void         RaisePostBackEvent(String eventArgument);
    }
}
```

Methods

RaisePostBackEvent	
Prototype	public void RaisePostBackEvent(String eventArgument);
Description	Enables a control to process the event produced by a form post back.
Parameters	*EventArgument* - Optional Event Argument

8.8 System.Web.UI.IPostBackDataHandler

The IPostBackDataHandler interface defines the contract that Controls that wish to automatically load posted back data should implement.

```
namespace System.Web.UI
{
    public interface IPostBackDataHandler
    {
        // Post-Back Data Binding Methods

        public bool         LoadPostData(HTTPValueCollection postData);
        public void         RaisePostDataChangedEvent();
    }
}
```

Methods

LoadPostData	
Prototype	public bool LoadPostData(HTTPValueCollection postData);
Description	Enables a control to load incoming form data posted from a client. Developers should return a "true" bool value if the post data causes the control state to change, and an appropriate "data change" callback needs to occur via a "RaisePostDataChangedEvent" call.
Parameters	*PostData* – Collection of all incoming HTTP Values.

RaisePostDataChangedEvent	
Prototype	public void RaisePostDataChangedEvent();
Description	Signals a control that it should notify any listeners that the state of the control has changed.

8.9 System.Web.UI.ITemplate

The ITemplate interface provides a factory definition for populating a control with children and property attributes from an in-line template within a page file.

```
namespace System.Web.UI
{
    public interface ITemplate
    {
        // Methods

        public void                Initialize(Control control);
    }
}
```

Methods

Initialize	
Name	public void Initialize(Control control);
Description	Iteratively populates a provided ControlCollection instance with a sub-hierarchy of child controls.
Parameters	*Control* – Control to be initialized from an in-line template.

8.10 System.Web.UI.TemplateAttribute

The TemplateAttribute class defines a custom attribute that can be used by controls that expose custom templates to indicate the Type of the control to be initialized at runtime. The WebForms parser will use this type information to code-generate a strongly typed member named "Container" in the resulting page class.

```
namespace System.Web.UI
{
    [AttributeUsage(AttributeTargets.Property)]
    public class TemplateAttribute : Attribute
    {
        // Properties

        public Type             ContainerType       { get; }

        // Constructor

        public                  TemplateAttribute(Type containerType);
    }
}
```

Properties

ContainerType	
Prototype	public Type ContainerType { get; }
Description	Indicates the COM+ Type of the control instance provided to an ITemplate member at runtime. Please review the "templates" section of this spec for more details.

8.11 System.Web.UI.INamingContainer

The INamingContainer interface is a marker interface that identifies a container control that scopes a new id namespace hierarchy within the tree.

```
namespace System.Web.UI
{
    public interface INamingContainer
    {
        // Marker Interface Only
    }
}
```

9. Todo

- Need to add more verbage about our bubbling support
- Need to add more verbage about DataBinding
- Add visual inheritance section

10. Change History

Date	Changes	By
02/14/00	Beta1 Document Created From Alpha Spec	ScottGu
02/24/00	Document Updated	ScottGu
02/29/00	Document Updated	ScottGu
03/07/00	Document Updated	ScottGu
03/08/00	Document Updated	ScottGu
03/10/00	Document Updated	ScottGu
03/11/00	Document Updated	ScottGu
04/19/00	Document Updated	ScottGu
05/01/00	Document Updated	ScottGu
05/07/00	Document Updated	ScottGu